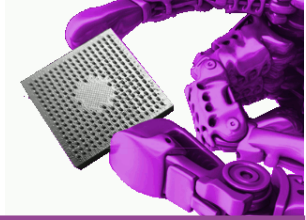


Computer Architecture

An embedded approach

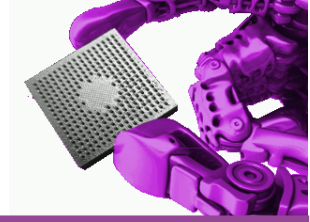


Module 3

CPU Basics

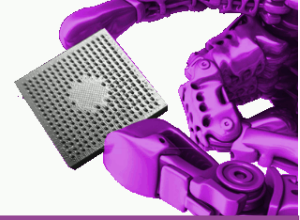
(making it work for you)

Contents



- 3.1 What is a computer?**
- 3.2 Making the computer work for you**
- 3.3 Instruction handling**
- 3.4 Data handling**
- 3.5 A top-down view**

Computer requirements

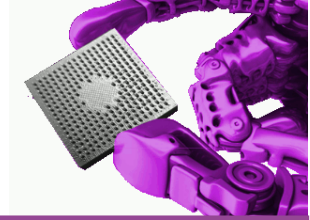


As you know, programs are stored in **machine code instructions** which are **fixed** (in most RISC devices such as ARM, PIC or MIPS) or **variable** length sequences of numbers (as in many CISC devices such as ADSP and Pentium).

Numbers stored in memory need to have a location that is accessible. Early computer designers termed the location an address, and this allows the CPU to select and access it. The most efficient way to do this has been for the CPU to output the address it wants, wait for the value in that address to be looked up, and to read in the value.

The address is output on an **address bus**, and the data is read on a data bus. Some early microcomputers had multiplexed data and address lines (to save on device pins). The number of wires in the address bus limits the number of address locations that can be accommodated.

Computer requirements

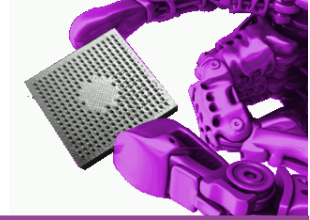


Once the instruction is inside the CPU it needs to be **decoded**, and then **executed**. Different units inside the CPU perform different tasks, so the data to be processed needs to be directed to the correct unit.

For this, there must be an internal CPU bus between the **instruction fetch and decode unit**, and the **processing unit**. Also a bus to collect the result of the processing unit, and place it somewhere.

In most modern CPUs, internal data is stored in **registers**, and this data is collected from registers by processing units, altered and returned to a register. So every processing unit must be connected to a bank of registers.....

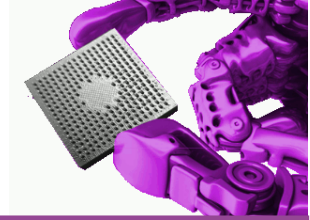
Computer requirements



However, in some processors, such as the ADSP2181, there is no bank of registers - there are instead specific registers associated with the input and output of each processing element.

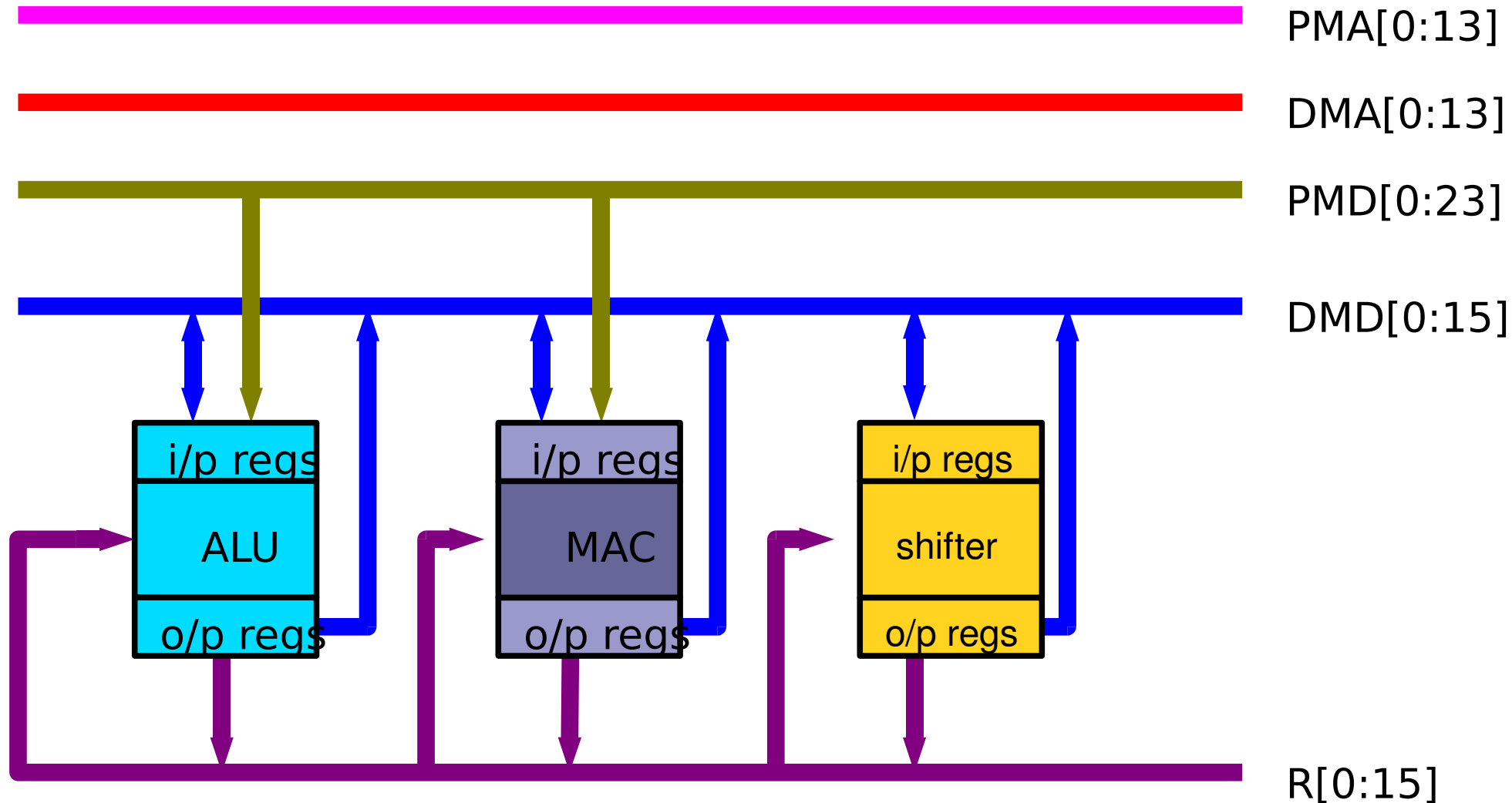
Each processing element is limited to receiving its input from only a few registers, and outputting to another small set. This means there are many internal buses. **Why?**

Consider a section of the internal design of the ADSP2181...



Computer requirements

Part of the ADSP2181 internal structure



PMA[0:13]

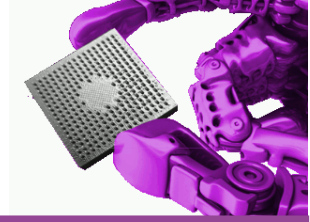
DMA[0:13]

PMD[0:23]

DMD[0:15]

R[0:15]

The need for memory



If a computer is just a box that **moves and manipulates data**, then there are three important things to consider:

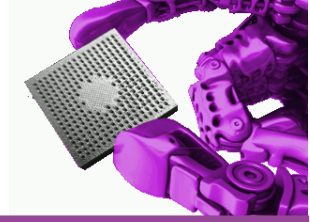
- How to move data from one place to another
- How to manipulate data
- How to store data in one or more places

Memory is storage, for both data to be processed, and for the program instructions that specify the movement and processing.

It can be volatile, non-volatile (more on that in Chapter 7)

It can be for long term storage or temporary storage.

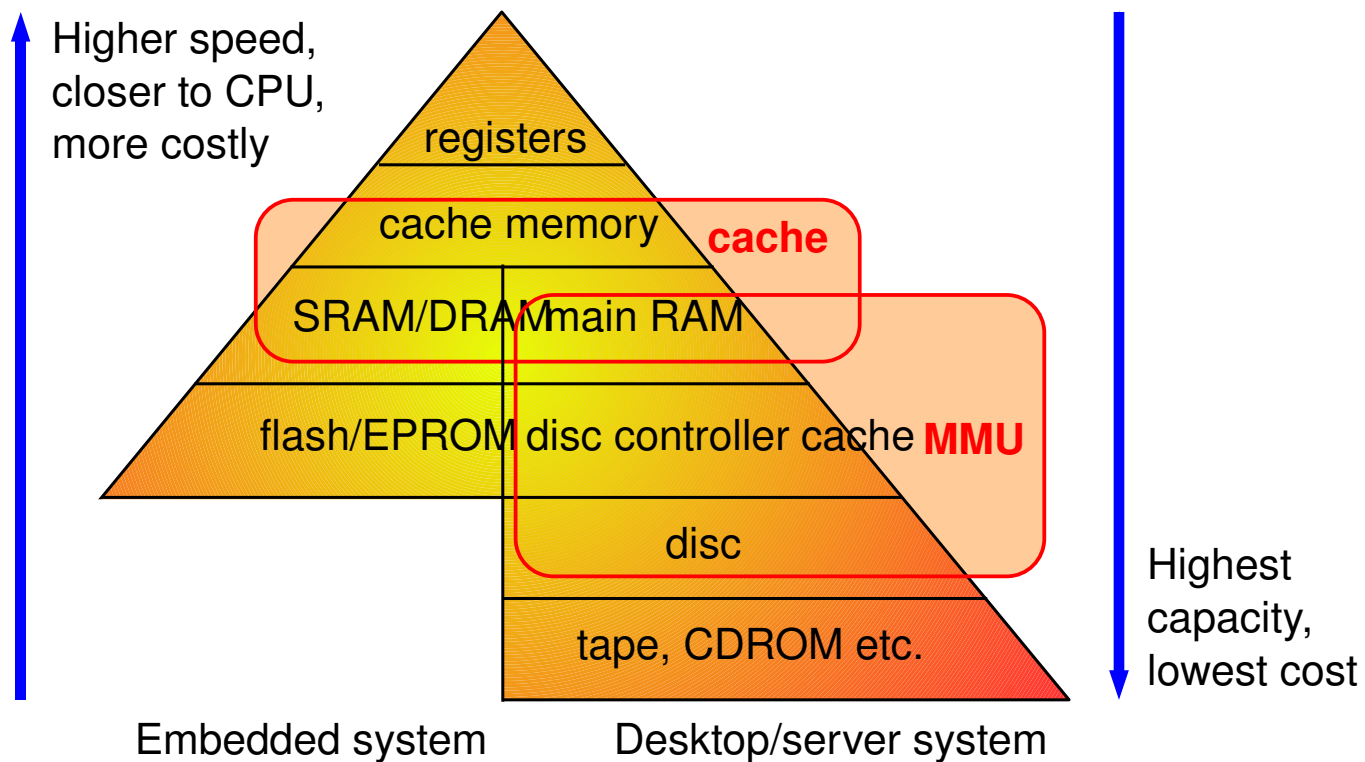
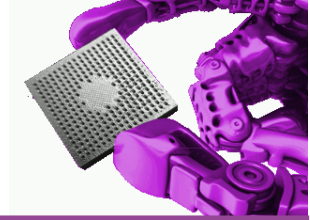
The need for memory



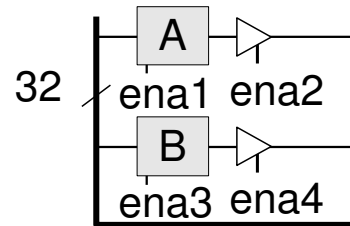
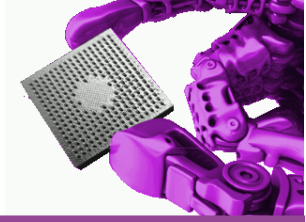
Some of the **characteristics** of memory:


- Cost
- Density (bytes per square or cubic centimetre)
- Power efficiency (nJ per write/read or second of storage time)
- Access speed (seek time, burst time, read time, write time)
- Access size (bit, byte, block or page)
- Volatility (when power is lost...)
- Reliability
- Management overhead (if any)

The hierarchy of memory

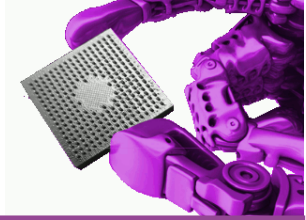


Transferring data

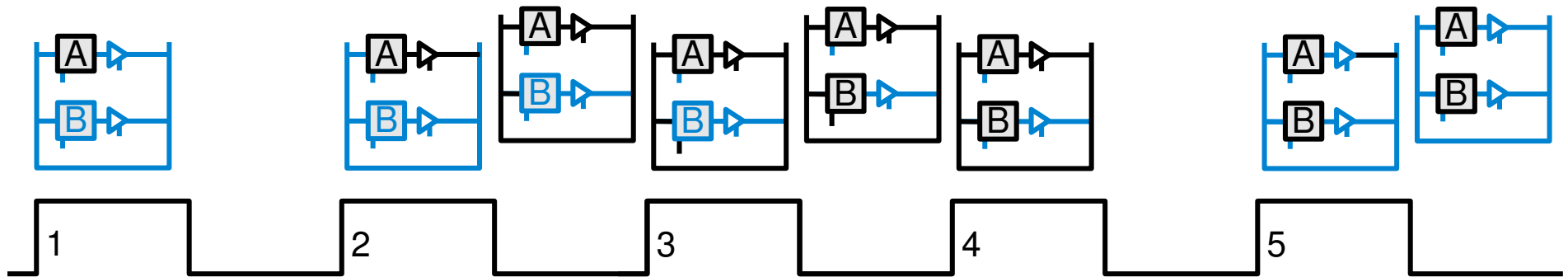


Buses transfer data to and from latches (or registers), and tristate buffers () arbitrate between these, all timed by a CPU clock, and all controlled by a **CPU control unit**

Transferring data

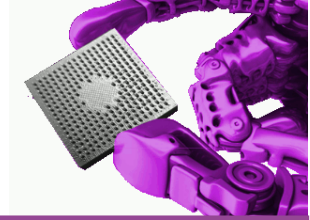


Timing is everything!



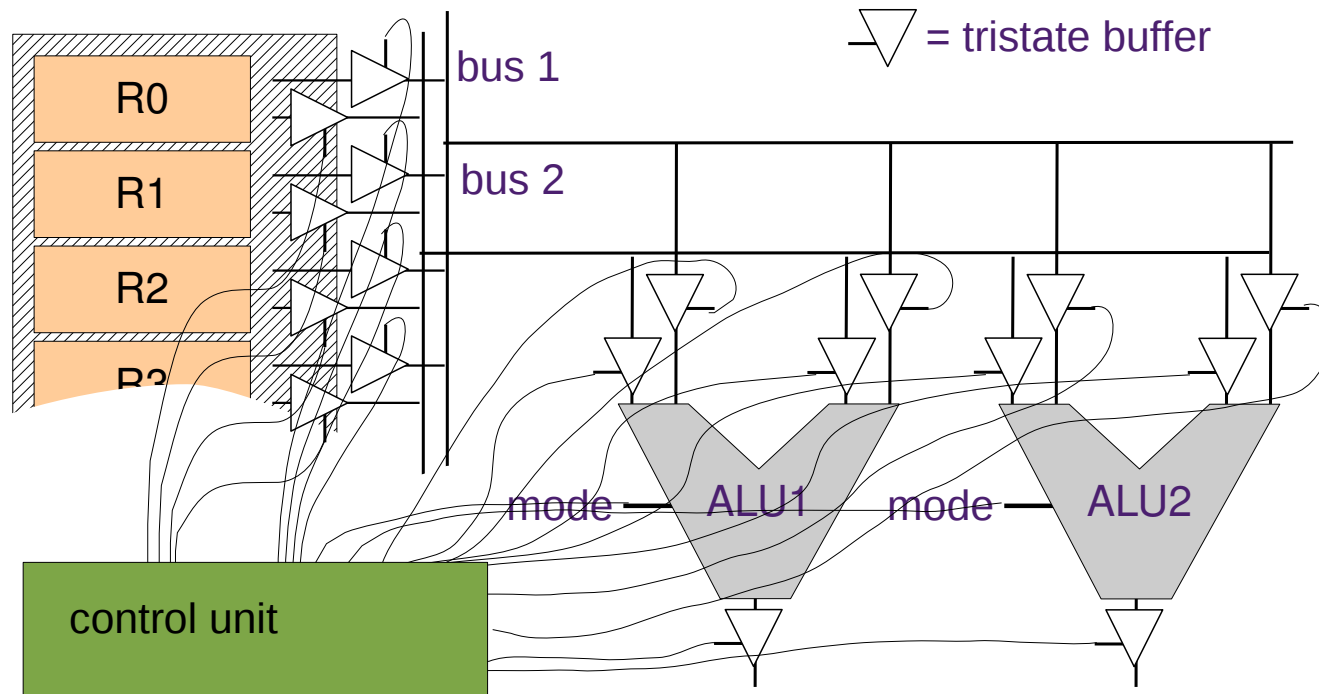
5 separate clock cycles are used to **enable** and **disable** buses, **tristates** and **latch inputs** in the correct sequence to **transfer data** from register A to register B.

Transferring data



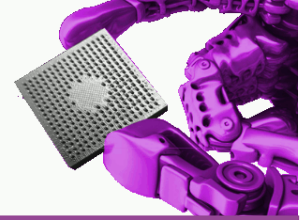
So who or what determines the sequence and timing?

The control unit!

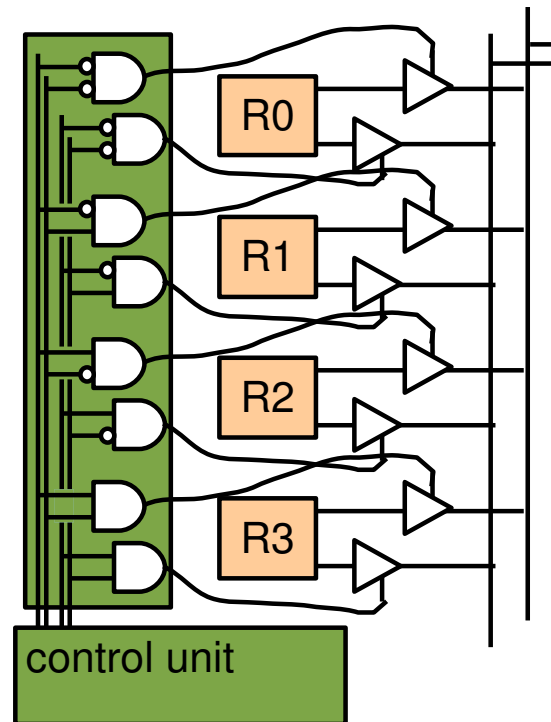


Yes – it's supposed to look like a complex mess...!

Transferring data

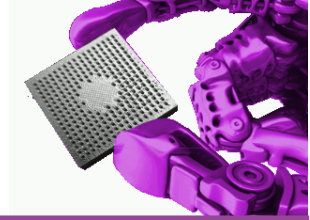


It is possible to have a neater control bus using multiplexing:

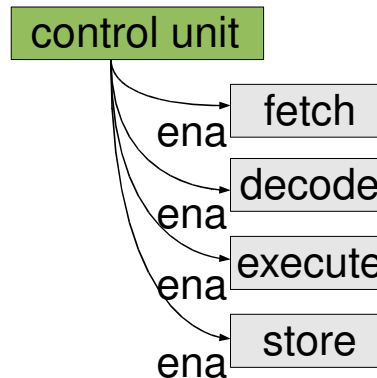


So we output **0000** to select R0, **0001** to select R1 and so on: in fact, this is the way the registers are encoded within the instruction word bits (see later).

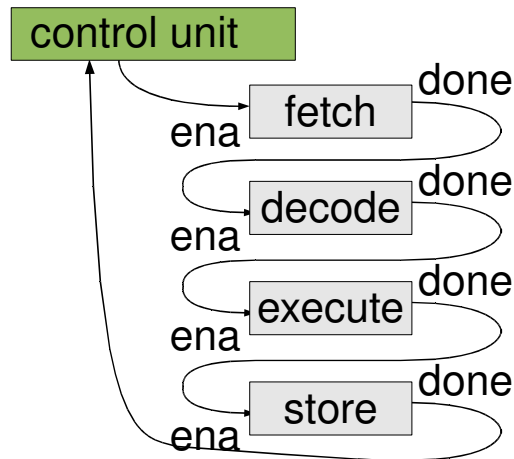
Transferring data



All examples so far have shown **centralised** control:

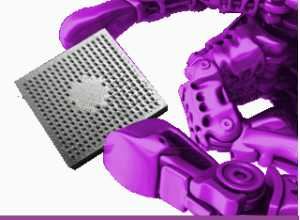


But **self-timed control** is also possible:

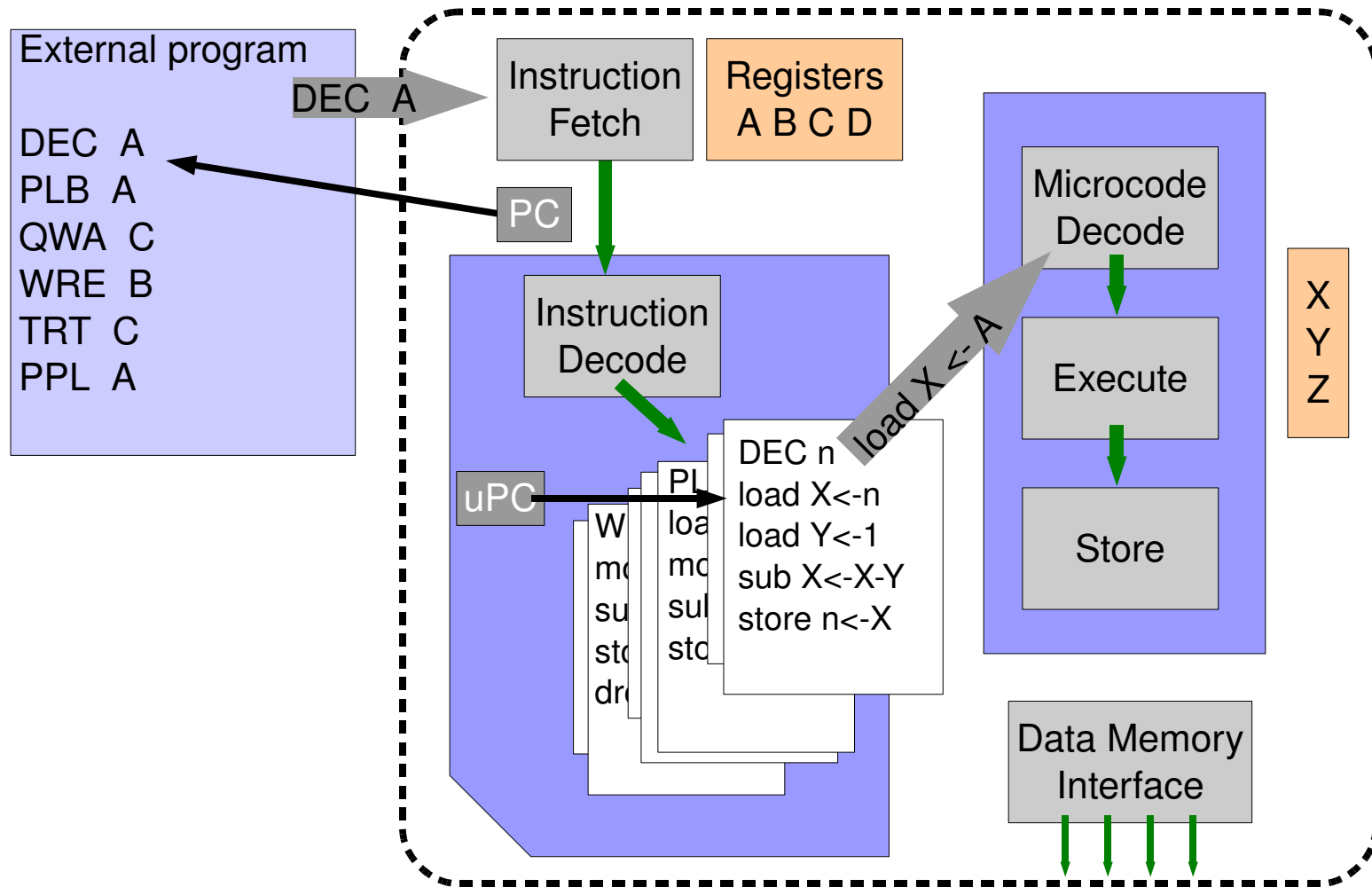


The control unit is much simpler (one output, one input!).
But **distributed control** is also possible.

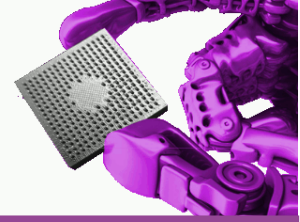
Transferring data



But for many years, **microcode** looked like a promising method:



Transferring data



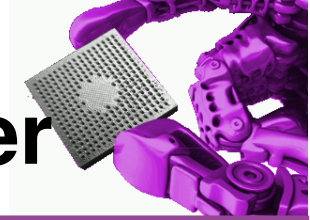
Microcode allows complex instructions to be broken into **microprograms**: sequences of simpler **microcode**.

This works well when:

1. **External memory is limited / expensive.**
2. The **bandwidth** from external memory into the CPU is a bottleneck.
3. One type of CPU needs to execute the instructions of another CPU (a type of **emulation**).

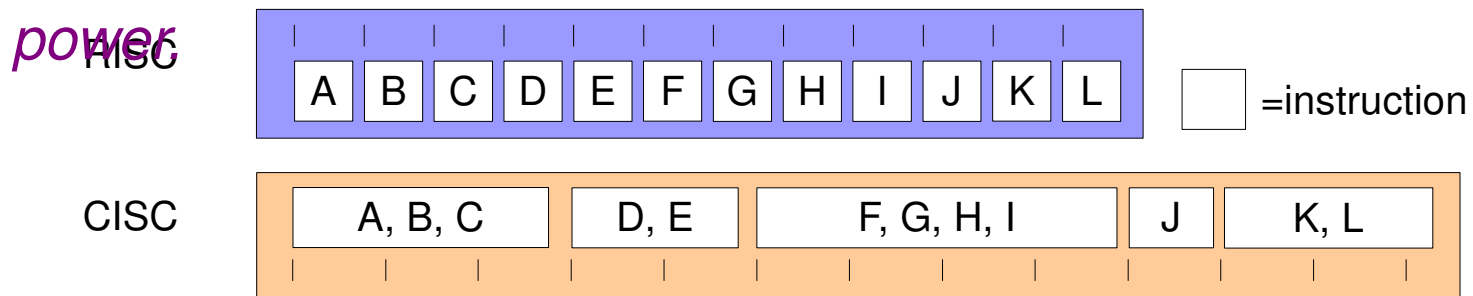
There is even something called **nanocode** (when each **microcode** instruction is itself a **nanoprogram!**).

Reduced Instruction Set Computer



Microcode instructions are typically very simple and fast to execute. In fact, we can make an entire CPU that handles only simple instructions, a **Reduced Instruction Set Computer (RISC)**.

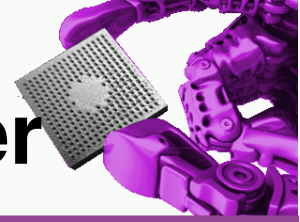
*Because the instructions are **simple**, they are **fast**, but we usually need more of them to perform a task. However the RISC CPU only has to handle simple instructions, so it becomes small, fast and low power*



In this example, 12 data operations are performed. The RISC device needs 12 instructions to do this. By contrast, the Complex Instruction Set Computer (CISC) has some powerful instructions that do several things together: it only needs 5 instructions to do the same job.

However the CISC instructions are slower, so it ends up taking longer to finish the 12 operations.

Reduced Instruction Set Computer



RISC

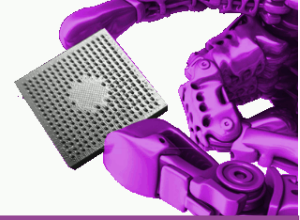
The **ARM** is the most famous **RISC architecture**, followed by the **SPARC**, **PowerPC** and **MIPS** designs.

CISC

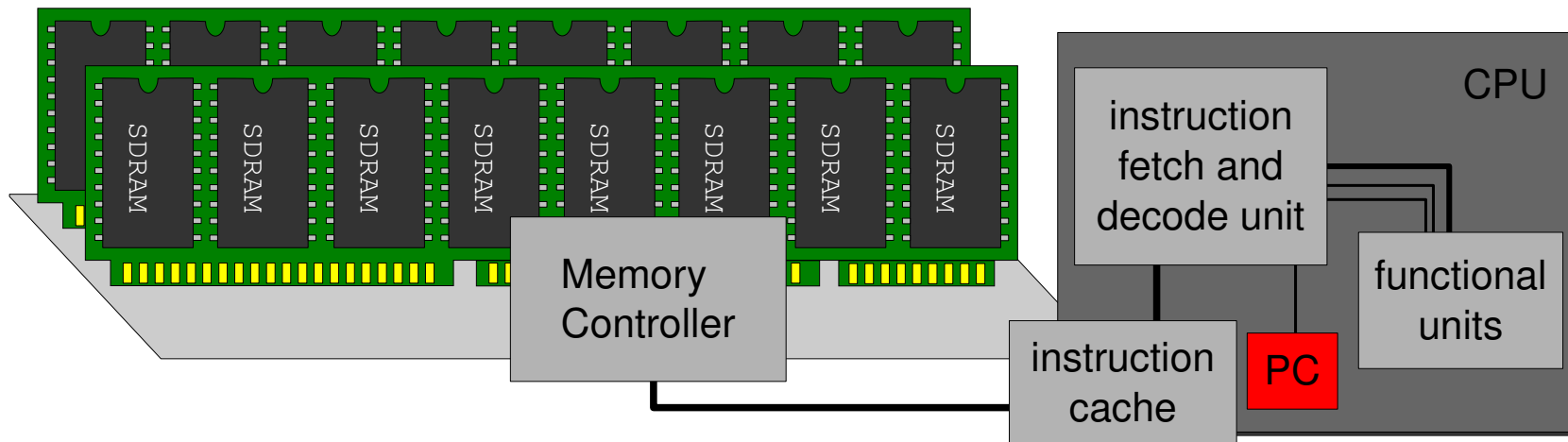
The **x86** series is the most famous **CISC architecture**, followed by **68000** and **IBM System/360** designs.

Interestingly, many of the modern incarnations of CISC design (including the Intel x86 series) actually use a RISC core plus a translation unit; in effect they use a RISC microprogram, and a RISC micro-engine to execute code.

Instruction Handling



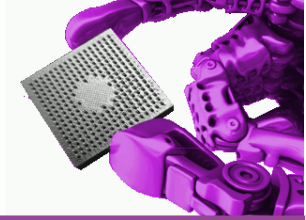
During operation, instructions from a program need to be fetched from program memory storage, and brought into the CPU for execution.



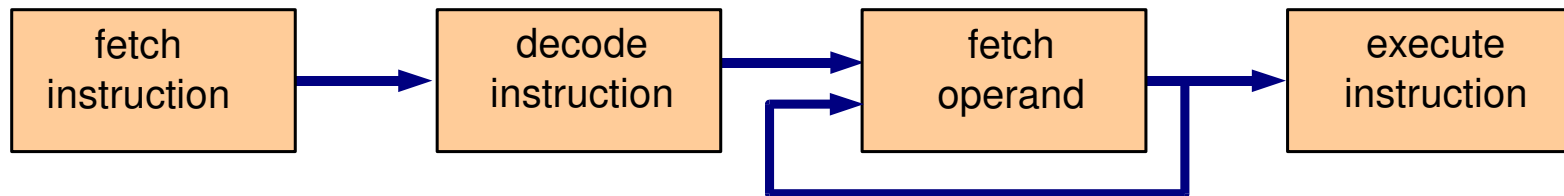
Inside the CPU, they are **decoded** – so the CPU can decide **how to execute them**: which **resources** such as **registers**, **buses** and **functional units** need to be involved, and in what order.

The **program counter (PC)** holds the address of the **next instruction** to be read in and executed.

Instruction Handling



The sequence of operations for program execution is:



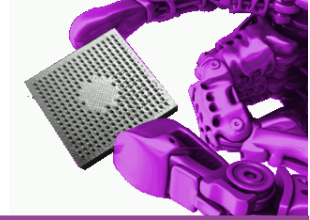
The loop around “**fetch operand**” is because there could be more than one operand (in fact, 0, 1, 2, 3... or more!).

In fact, the diagram should continue after the “**execute**” stage:

- Update condition code flags
- Store result

Usually the PC is automatically incremented by 1 right after instruction fetch, so it points to the next instruction.

Instruction Handling

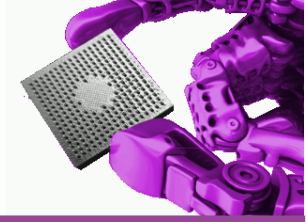


How is **branching** performed?

B target

1. The **target address** is part of the **instruction word** – an **immediate operand**.
2. We simply **store** the **target address** into the **PC register**.

Instruction Handling



Most processors use a conditional branch to form if-then-else, for-next, do-while and case-switch constructs:

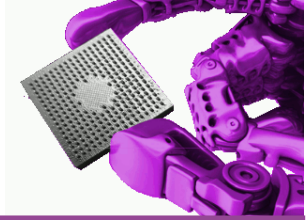
Bxx target

Where xx is a condition code, like this, which branches is the condition flags (set by some previous instruction) resulted in an answer that was non-zero:

BNZ target

However the ARM is cleverer, it can use a condition code with every instruction (see Box 3.3 on page 87)!

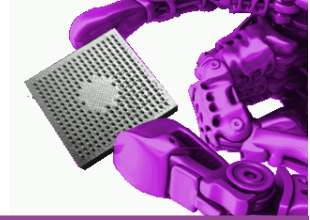
Instruction Handling



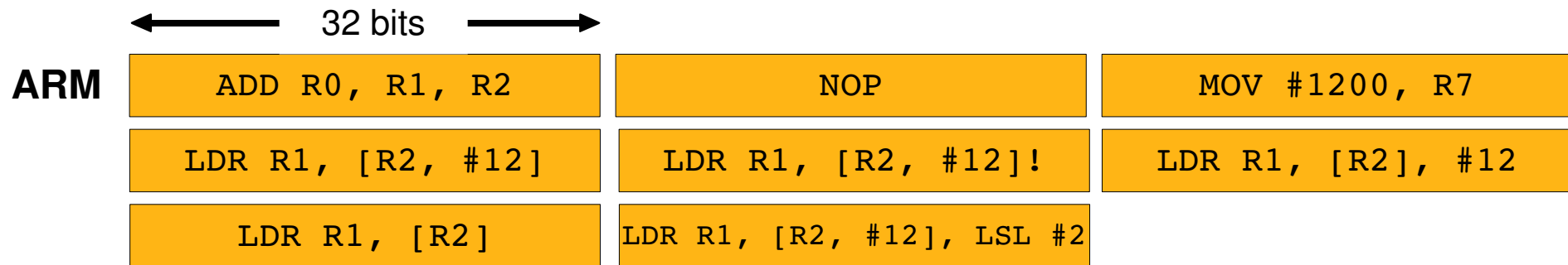
Here are the **condition codes** used by the ARM CPU (and the bits in the instruction word that encode them):

Bits	Code	Meaning	Conditional on
0000	EQ	equal	Z=1
0001	NE	not equal	Z=0
0010	CS	carry set	C=1
0011	CC	carry clear	C=0
0100	MI	minus	N=1
0101	PL	plus	N=0
0110	VS	overflow set	V=1
0111	VC	overflow clear	V=0
1000	HI	higher	C=1, Z=0
1001	LS	lower or same	C=0, Z=1
1010	GE	greater or equal	N=V
1011	LT	less than	N=~V
1100	GT	greater than	N=V, Z=0
1101	LE	less than or equal	(N=~V) or Z=1
1110	AL	always	-
1111	NV	never	-

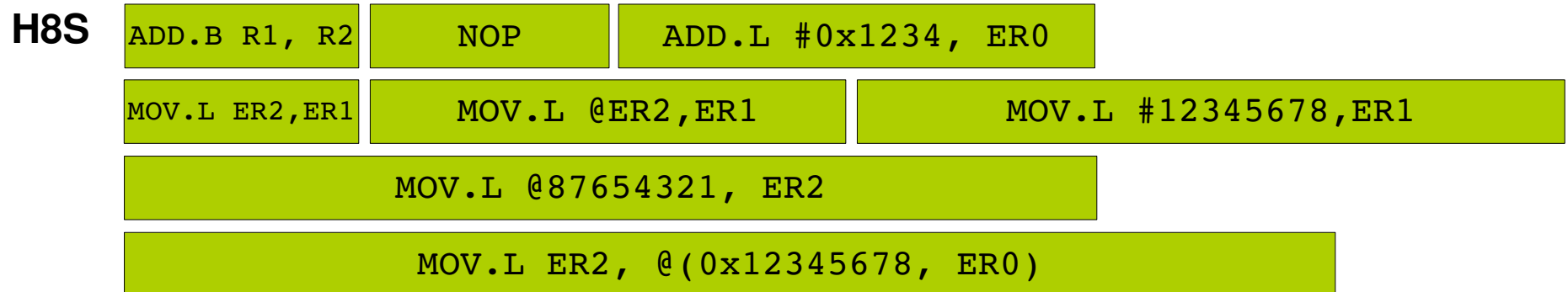
Flag names: **Z** = zero, **C** = carry, **V** = overflow, **N** = negative



Variable-length Instructions

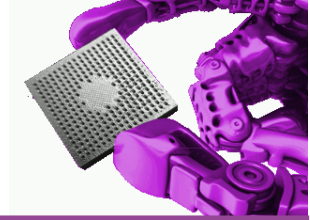


In most RISC processors, all **instructions** are of **equal length**, and a lot can be accomplished with those fixed number of bits! But this usually goes hand-in-hand with a **load-store architecture**.



Some processors, particularly CISC devices, have **variable length instructions**, depending upon what information needs to be incorporated into the instruction word.

Variable-length Instructions



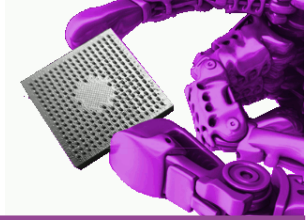
For processors with variable-length instructions, **Huffman encoding** can be used to improve processor efficiency.

This is based on the procedure of **reducing the size of the most common instructions**, and increasing the size of the least common instructions. This provides an overall size improvement.

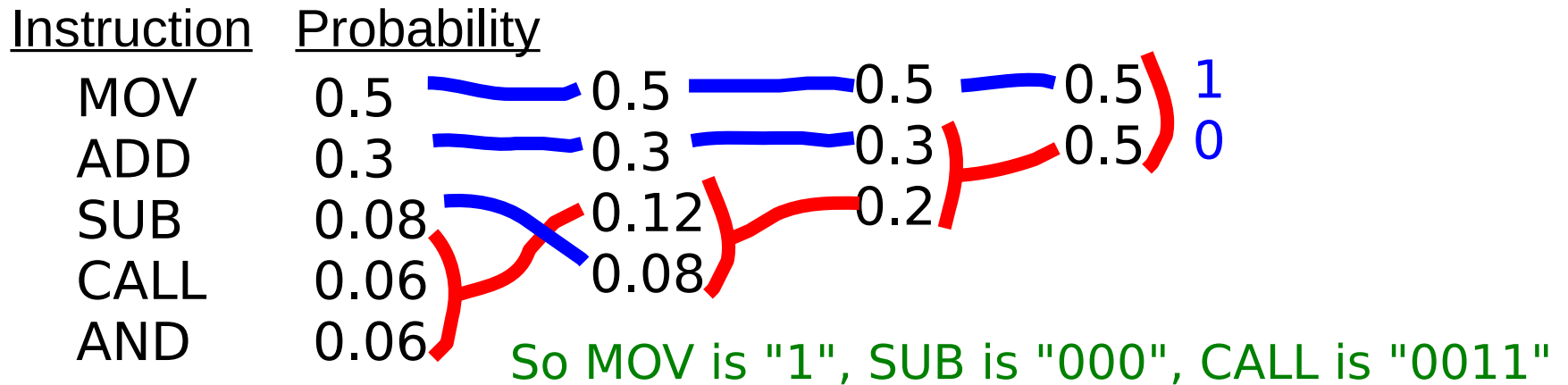
For n instructions, the probability of those instructions occurring is found. The size of the encoded word used to represent instructions is inversely proportional to their probability (i.e. more common instructions have shorter instruction words).

Note that in the real world, some applications may have very different instruction statistics compared to the average...

Variable-length Instructions



Huffman example for 5 instructions:



Order in terms of probability working from left to right.

Combine lowest two probabilities in the first column (add their probabilities together) and then re-order the remainder in the next column.

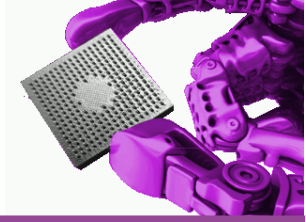
Combine the two lowest probabilities in this column, then reorder the remainder into another column.

Continue until only two are left.

Then scan back through the tree from right to left.

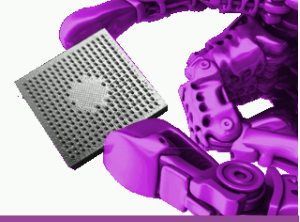
At each column, call the upper path "1" and the lower path "0".

Addressing modes



- 1 data register direct
- 2 address register direct
- 3 immediate addressing
- 4 absolute addressing
- 5 address register indirect
- 6 address register indirect with displacement
- 7 address register indirect with index and displacement
- 8 address register indirect with postincrement
- 9 address register indirect with predecrement

Addressing modes



Addressing modes: which are most efficient?

Here are some examples of using main memory for storage;

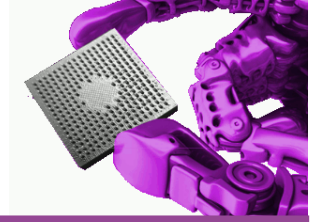
ADD A, B, C $A=B+C$ CPU needs to read 3 memory locations,
and instruction must encode these 3 locations.

ADD A, B $A=A+B$ CPU needs to read 2 memory locations,
and instruction must encode these 2 locations.

ADD B $ACC=B+ACC$ CPU reads 1 memory location,
and instruction only encodes 1 location.

Is it possible to require no main memory locations?

Addressing modes



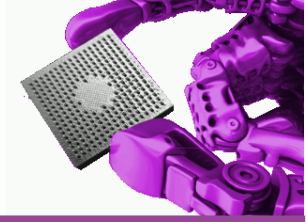
Addressing modes: which are most efficient?

ADD R0, R1 $R0=R0+R1$ CPU needs to access 2 registers, and instruction encodes these 2 registers. But if there are 15 registers, we need 4 bits to encode each register...

ADD CPU pops the top two stack entries, adds them and pushes the result onto the stack. This needs to access the stack, but the instruction does not need to encode any variable locations.

So how can we ensure that the stack holds the correct values?

Stack-based computation



Reverse Polish Notation (RPN):

Normally we use *infix* notation: $a + b/c$

With infix, there is a fixed precedence of operators (that can be overridden using parentheses), but RPN is a *postfix* operation, where the order of the equation defines the precedence.

When using a stack, scan from L to R, pushing values onto stack.

infix

$a \times b$

$a + b / c$

$(a - b) \times c$

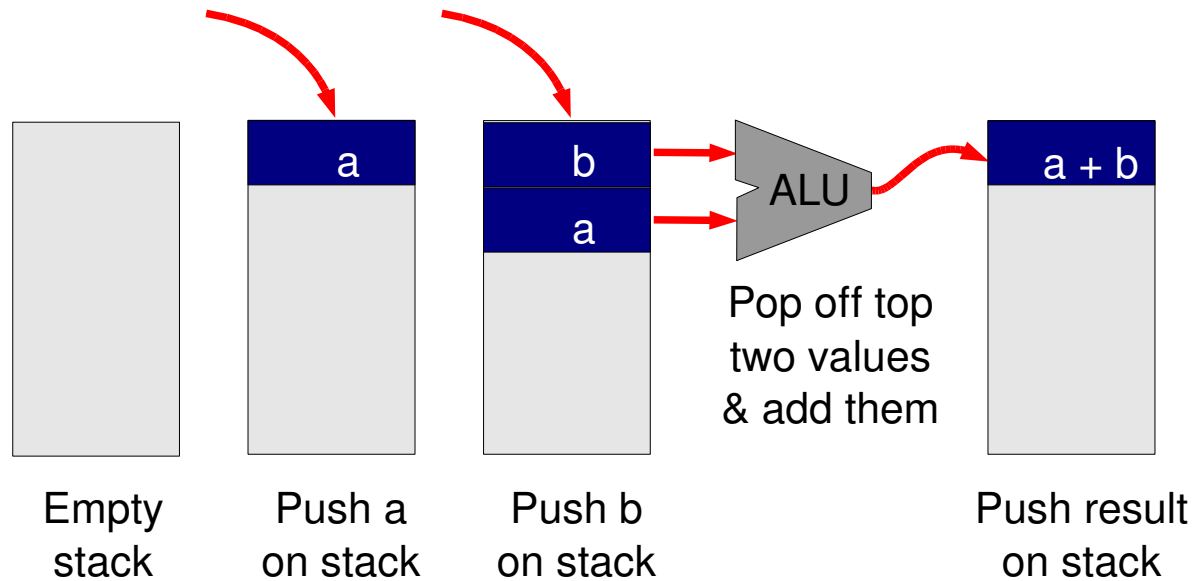
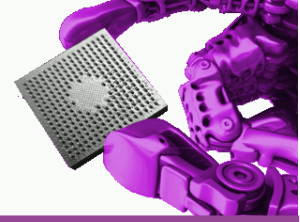
postfix

$ab \times$

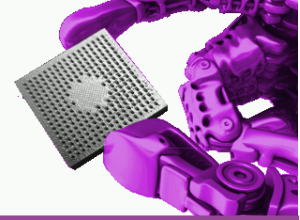
$abc / +$

$ab - c \times$

Stack-based computation



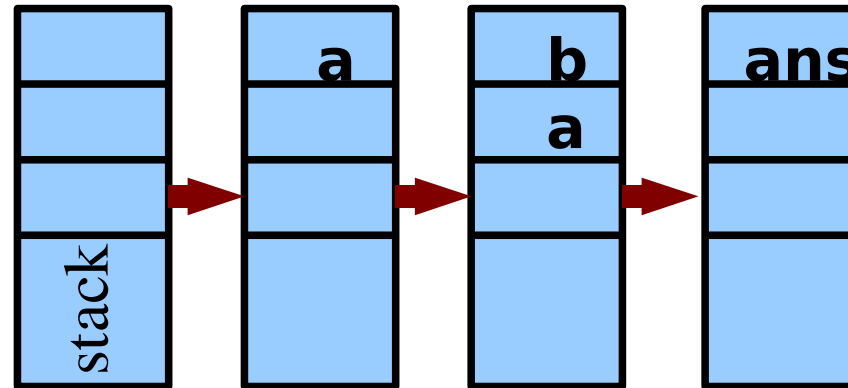
Stack-based computation



Reverse Polish Notation is useful for formatting stack operations

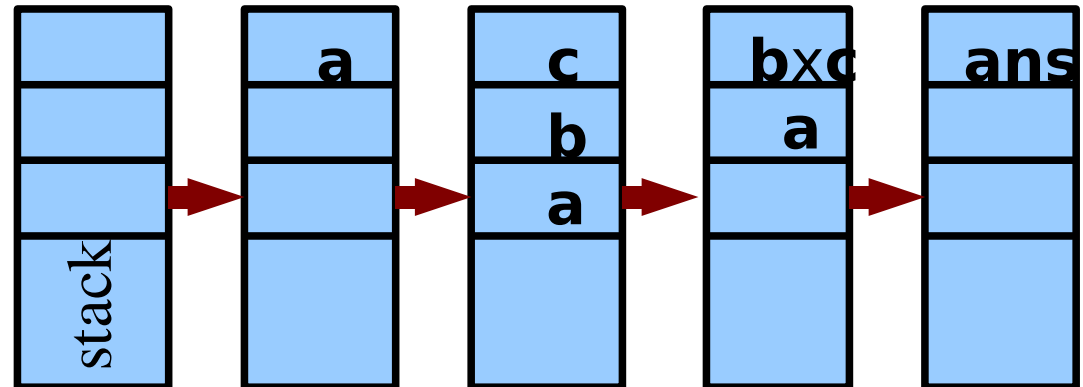
Infix
a+b

postfix
ab+



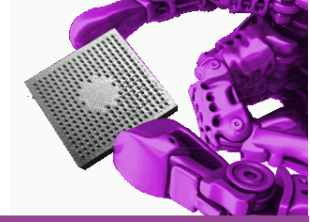
a+b/c

abc/+



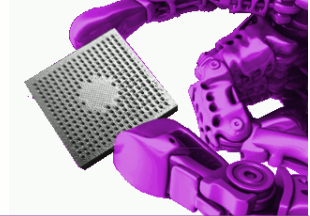
filling and emptying of the stack: each operator takes its argument from, and places its result on the stack.

Stack-based computation



We will learn more about stack-based computation later when we start working on TinyCPU...

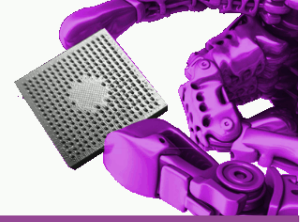
Computer performance



How do I measure how fast is my computer?

- How many GHz (or MHz) does my processor clock at?
 - How fast is the FSB (front side bus)?
 - How fast is the cache memory (and how much is there)?
 - How fast is my main memory (and how much is there)?
 - The CPU width (is a 128-bit CPU always faster than a 32-bit one?)
 - How powerful are my CPU instructions?
-
- How many MIPS (million instructions per second)?
 - How many cycles per instruction?
 - How many instructions per cycle?
-
- How many dhrystones/whetstones?
 - How many SPECint/SPECfp's?

Computer performance



The answer is that all of these can be used, and have been used to measure computer speed.

But be careful what you use to make a judgement!

Read Section 3.5 (pages 109 to 115): maybe you just have to try it, running your program or application, to find out...